Extracting Herbrand trees from Coq

Lionel Rieg

Technical report
Extracting Herbrand trees from Coq

Lionel Rieg
LIP, ENS de Lyon, France
lionel.rieg@ens-lyon.fr

December 2011

Abstract

Software certification aims at proving the correctness of programs but in many cases, the use of external libraries allows only a conditional proof: it depends on the assumption that the libraries meet their specifications. In particular, a bug in these libraries might still impact the certified program. In this case, the difficulty that arises is to isolate the defective library function and provide a counter-example. In this paper, we show that this problem can be logically formalized as the construction of a Herbrand tree for a contradictory universal theory and address it. The solution we propose is based on a proof of Herbrand's theorem in the proof assistant Coq. Classical program extraction using Krivine's classical realizability then translates this proof into a certified program that computes Herbrand trees. Using this tree and calls to the library functions, we are able to determine which function is defective and explicitly produce a counter-example to its specification.

1 Introduction

Software certification has become in the last decade one of the major applications of computer assisted-proofs. This technique decomposes in two stages: the first one is a formalization step where we express the program properties we want as logical formulæ—call them V— whereas the second one focuses on proving them, which is usually done within a proof assistant. Provided we trust it[1], the assisted proof ensures that the program satisfies its specification V and thus behaves as expected.

However, in real life, programs may depend on external components, and in many situations we do not want (lack of time or money) or we are not able to certify them. This is typically the case when we do not have access to their implementation, *e.g.* a piece of proprietary software or a hardware component (processor, data acquisition device). In this situation, the correctness of the whole program (such as expressed by V) depends on correctness assumptions—call them U—about the external components. In practice, the formula we must prove is the implication $U \Rightarrow V$.

Having a proof of $U \Rightarrow V$ gives us no warranty on the correctness of the program with respect to the specification V, it simply means that when a bug report contradicts the specification V, we know that at least one of the components does not meet its specification.

1.1 Motivation: the Static Debugger

In the case where the certified program runs into a bug (according to V), traditional proof assistants provide no tool to track down the bug through the formal proof of $U \Rightarrow V$ to find out which external component is defective (according to U) and for which set of parameters—the *counter-example*.

The problem we are facing is the problem of *experimental modus tollens*, that can be summarized by the following pseudo-rule.

$$\frac{\vdash U \Rightarrow V \qquad \nvDash V}{\nvDash U}\ \text{Experimental Modus Tollens}$$

[1] The confidence that one should have in a proof assistant is not the topic of this paper so we will simply assume it.

This problem is the following: given a formal proof of the implication $U \Rightarrow V$ and a counter-example to the property V, how can we find a concrete counter-example to the property U? Here, the difficulty comes from the fact that we need to combine objects of two different natures: a formal proof ($\vdash U \Rightarrow V$) and an experimental evidence of misbehavior ($\not\models V$).

Notice that we are using the symbol $\not\models$ not in its usual meaning: here we do not only want a *counter-model* (which is obvious, a counter-model for V also works for U) but rather a *counter-example*, *i.e.* a given set of parameters falsifying U. For that, the specifications U and V must be universal, namely Π_1 formulæ (or conjunctions thereof), otherwise the very notion of a counter-example does not make sense. In practice, this restriction is inoffensive because Π_1 formulæ are enough to express most behavioral properties of programs[2].

In [Miq09b, Miq11], Miquel proposes a method to solve this problem using the tools of classical realizability [Kri09]. The advantage of this method is that it also works when the proof of $U \Rightarrow V$ is classical, using the interpretation of classical reasoning in terms of control operators such as `callcc` [Gri90].

The aim of this paper is to present an alternative solution to this problem based on a formalization of Herbrand's theorem in Coq. To present this solution, we first need to dwell on some details of the experimental modus tollens.

1.2 Logical Formalization

As we have already said, the formulæ U and V must be universal. But they are not arithmetical because they contain function symbols describing external components and predicate symbols expressing their properties. Formally, we simply work in a first-order signature containing these function and predicate symbols. (To ensure the concrete effectiveness of the method, we need that these function and predicate symbols actually correspond to components and properties that can be concretely tested.)

Instead of working with the full rule of experimental modus tollens, we shall work with the particular case where V is the false formula.

$$\frac{U \vdash \bot}{\not\models U}\ \text{Experimental Effectiveness}$$

This rule expresses that from a contradictory theory, we can deduce a counter-example.

Although the problem of experimental effectiveness is a particular case of the problem of experimental modus tollens, they are in fact equivalent. The transformation between these two pseudo-inference rules goes as follow: if we denote by V_0 the particular instance of V that was experimentally falsified, by eliminating the $\forall$ connectives of V, we can build a proof of $\vdash V \Rightarrow V_0$. Combining it with the given proof of $\vdash U \Rightarrow V$, we make a proof of $U \wedge \neg V_0 \vdash \bot$. Using experimental effectiveness, we get a counter-example for $U \wedge \neg V_0$. Since we know that V_0 is falsified, it is a counter-example for U.

1.3 Abstracting tests

The solution presented by Miquel consists in combining the classical λ-term extracted from the classical proof of $U \vdash \bot$ with a wrapper embedding extra instructions performing the tests associated with the predicate symbols of the first-order signature. In practice, the experimental tests associated with the external predicates may be expensive to perform and we would like to remove them. Nevertheless, we cannot completely avoid them if we want to find a counter-example, because they appear in the specification U we want to falsify and because tests are the only way to get information about them.

One intermediate solution is to abstract over these tests and consider all possible interpretations for the external predicates. Instead of a single counter-example, this will lead to a *family* of counter-examples organized in the form of a Binary Decision Diagram (BDD), whose internal nodes will represent the atomic experiments that must be performed in order to reach a counter-example at a leaf. From the point of view of logic, this BDD is nothing but a Herbrand tree falsifying the formula U.

Formally, a Herbrand tree is a finite binary tree whose inner nodes are labeled with atomic formulæ. Intuitively, each branch of such a tree is a partial interpretation of the atomic formulæ which is sufficient to determine a counter-example (to the universal theory U) that is placed at the corresponding leaf. Given a Herbrand tree and components testing the truth of atomic formulæ, it is easy to extract the

[2]Program properties are usually of the form "for all possible inputs, there is a specific relation between an input and the corresponding program output"

desired counter-example using the tree as a BDD: go down in the tree by testing encountered atomic formulæ and entering the sub-tree corresponding to the result of the test, until reaching a leaf.

1.4 Extracting Herbrand trees from a proof of Herbrand's theorem

It is well-known in logic that any universal formula U that is contradictory has a Herbrand tree.

Theorem 1. — *If U is a universal inconsistent theory, then U has a Herbrand tree.*

This theorem exactly solves our problem: find a counter-example (abstracted over the interpretations of atomic formulæ) for a contradictory universal theory. The only trouble is that we simply have the existence of Herbrand trees whereas we want an explicit procedure to build them. To get round this difficulty, we propose to build a proof of Herbrand's theorem in a proof-assistant and to extract it to a program computing Herbrand trees. Since the usual proof of Herbrand's theorem is classical, we need a classical extraction mechanism. This currently only exists for the Coq proof assistant [CDT10] as the classical extraction module kextraction [Miq09a] based on Krivine's classical realizability [Kri09].

The methodology we propose in this paper is to formalize a (classical) proof of Herbrand's theorem in Coq, apply it to a proof of contradiction of some universal theory U and extract the resulting theorem (expressing the existence of a Herbrand tree for U) through classical extraction. The execution of this extracted term will finally lead to the tree we look for.

1.5 Outline of the paper

The most difficult part of the presented methodology is the proof of Herbrand's theorem in Coq. It will be the topic of the first two sections, the first one focusing on a reformulation of Herbrand's theorem in a suitable statement for Coq manipulation and the second one dwelling in the details of the formalized proof. The next section explains how to connect the formalized proof of Herbrand's theorem with a contradiction proof of a universal theory in order to get a theorem expressing the existence of a Herbrand tree. The fifth section focuses on the last steps of our methodology: classical extraction in Coq and evaluation of the extracted term. Finally, the sixth section presents an completely different approach to the problem of building Herbrand trees, using a customized classical realizability.

2 Expressing Herbrand's theorem in Coq

2.1 Making Herbrand's theorem Coq friendly

Instead of using the proof-theoretic assumption of having a proof of contradiction $U \vdash \bot$ for a theory U, we rather use its model-theoretic counterpart, *i.e.* a proof of its inconsistency $\vdash \forall \mathscr{M}, \mathscr{M} \not\models U$. Nevertheless, this new statement is not very convenient to use in Coq because it requires to express model theory in Coq which we would rather avoid. In this section, we shall explain how to transform the hypothesis of Herbrand's theorem into a more suitable statement for manipulation with Coq.

First of all, because U is a finite universal theory, we can write it as

$$U \equiv \bigwedge_{j=1}^{n} \forall x_1 \dots \forall x_{k_i}, C_j(x_1, \dots, x_{k_i})$$

where n is the size of the theory and the C_i are quantifier-free formulæ with k_i variables built from atomic formulæ with the usual logical connectives. The hypothesis $\forall \mathscr{M}, \mathscr{M} \not\models U$ is then classically equivalent to

$$\forall \mathscr{M}, \exists i\ \exists \overrightarrow{v} \in |\mathscr{M}|^{k_i},\ \mathscr{M} \not\models C_i(\overrightarrow{v})$$

where $|\mathscr{M}|$ is the carrier of the interpretation $\mathscr{M}$. In order to have only one existential quantifier, we merge the parameters i and $\overrightarrow{v}$ by introducing the dependent sum

$$\texttt{index} = \sum_{i=1}^{n} |\mathscr{M}|^{k_i} = \{(i, \overrightarrow{v}) | \overrightarrow{v} \in |\mathscr{M}|^{k_i}\}$$

so that we can rewrite our hypothesis as

$$\forall \mathscr{M}, \exists i : \texttt{index},\ \mathscr{M} \not\models C_{\pi_1(i)}(\pi_2(i)).$$

At this point, we observe that we are not fully using the C_j but only their ground instances $C_j(\overrightarrow{v})$. So instead of manipulating a full syntax containing terms and atomic formulæ built upon terms and arbitrary predicates, we can simply abstract the syntax by an abstract data type `atom` representing atomic formulæ. With such an `atom` data type, we build quantifier-free formulæ as the elements of the Boolean algebra generated by the atoms, a data type we will call `compound`.

$$c, d : \texttt{compound} \quad ::= \quad a \quad | \quad c \wedge d \quad | \quad c \vee d \quad | \quad \neg c$$

where $a \in \texttt{atom}$. We express the dependency on the `index` i of the `compound` associated with $C_{\pi_1(i)}(\pi_2(i))$ by a function $\mathrm{Th} : \texttt{index} \to \texttt{compound}$. Therefore, we now have the following representation in Coq:

$$\forall \mathscr{M}, \exists i : \texttt{index}, \mathscr{M} \not\models \mathrm{Th}\, i$$

where `atom` abstracts the signature and `index` and Th abstract the universal theory U.

Finally, since an interpretation $\mathscr{M}$ is only defined by its values over the atoms (we no longer have terms, so we are back to propositional calculus), we can take it to be a function from `atom` to `Prop`. Extending it in the straightforward way to `compound` (with a function $\mathrm{eval} : (\texttt{atom} \to \texttt{Prop}) \to \texttt{compound} \to \texttt{Prop}$) and generalizing over `atom`, `index` and the theory (*i.e.* over Th), we get this final statement as the hypothesis of Herbrand's theorem:

$$\begin{array}{ll} \forall \texttt{atom}, & \\ \forall \texttt{index}, & \forall\, \mathrm{Th} : \texttt{index} \to \texttt{compound}, \\ & \forall val : \texttt{atom} \to \texttt{Prop}, \neg(\forall i : \texttt{index}, \mathrm{eval}\, val\, (\mathrm{Th}\, i)) \end{array}$$

We will now illustrate this transformation on two simple examples.

2.2 The White Crow example

This first example is built upon a signature containing three unary predicate symbols Crow, Black and White, whose intended meaning is obvious. The individuals are a countable number of birds represented by integers.
Our ornithology says:

1. every crow is black, $\forall n, \neg\, \mathrm{Crow}\, n \vee \mathrm{Black}\, n$
2. a bird cannot be both black and white; $\forall n, \neg(\mathrm{Black}\, n \wedge \mathrm{White}\, n)$

whereas experimental observations have concluded that:

3. the bird 42 is a crow, $\mathrm{Crow}\, 42$
4. the bird 42 is white; $\mathrm{White}\, 42$

so that the White Crow theory will be

$$\big(\forall n, \neg\, \mathrm{Crow}\, n \vee \mathrm{Black}\, n\big) \wedge \big(\forall n, \neg(\mathrm{Black}\, n \wedge \mathrm{White}\, n)\big) \wedge \big(\,\mathrm{Crow}\, 42\big) \wedge \big(\,\mathrm{White}\, 42\big).$$

Atoms represent atomic formulæ of the signature. In this case, atomic formulæ are threefold and ranges over the set $\{\mathrm{Crow}(n), \mathrm{Black}(n), \mathrm{White}(n) \mid n \in \mathbb{N}\}$. Therefore, the set of atoms can be taken to be $\{C_n, B_n, W_n \mid n \in \mathbb{N}\}$ where the C_n, B_n and W_n are constants. The set of indices is the disjoint union of the set of parameters of the axioms of the theory:

$$\{(1, n) \mid n \in \mathbb{N}\} \cup \{(2, n) \mid n \in \mathbb{N}\} \cup \{(3, \emptyset)\} \cup \{(4, \emptyset)\}$$

Finally, Th translates an `index` into the corresponding `compound`:

$$\mathrm{Th}\, i = \begin{cases} \neg C_n \vee B_n & \text{if } i = (1, n) \\ \neg(B_n \wedge W_n) & \text{if } i = (2, n) \\ C_{42} & \text{if } i = (3, \emptyset) \\ W_{42} & \text{if } i = (4, \emptyset) \end{cases}$$

2.3 The pseudo-induction example

The second example is even simpler: from one constant symbol a, a unique function symbol f of arity 1 and one unary predicate symbol P, we build the following theory:

$$\big(\forall x, P(x) \Rightarrow P(f(x))\big) \wedge P(a) \wedge \neg P(f(f(a))).$$

Since we have only one predicate symbol, the set of atoms is $\{P(t) \mid t \text{ is a term}\} = \{P(f^n(a)) \mid n \in \mathbb{N}\}$ written $\{a_n \mid n \in \mathbb{N}\}$. The set of indices is $\{(1,t) \mid t \text{ is a term}\} \cup \{(2,\emptyset)\} \cup \{(3,\emptyset)\}$ and is more conveniently expressed as $\{(1,n) \mid n \in \mathbb{N}\} \cup \{2,3\}$. The theory is then described by the function

$$\operatorname{Th} i = \begin{cases} \neg a_n \vee a_{n+1} & \text{if } i = (1,n) \\ a_0 & \text{if } i = 2 \\ \neg a_3 & \text{if } i = 3 \end{cases}$$

3 A proof of Herbrand's theorem in Coq

Let us first recall the usual proof of Herbrand's theorem.

Theorem 2. *If U is a universal inconsistent theory, then U has a Herbrand tree.*

Proof. Let us fix an enumeration $(a_n)_{n\in\mathbb{N}}$ of the atomic formulæ built upon the signature of U. We consider the infinite binary tree enumerating these atoms in the order given by $(a_n)_{n\in\mathbb{N}}$: this tree has 2^n nodes at depth n each labeled by a_n. By definition, any infinite branch contains all the atoms and can be seen as an interpretation of the signature of U. Pick one infinite branch. Since U is inconsistent, this interpretation is not a model of U and thus induces a contradiction in U. By compactness, only a finite number of atomic formulæ are used to reach this contradiction, thus we can cut this infinite branch at finite depth while keeping the contradiction and any interpretation extending this partial interpretation will contain the same contradiction. We conclude using Kőnig's lemma to get a finite tree. □

This proof is classical because we use Kőnig's lemma. Since we want to extract the proof we will build, we use the proof assistant Coq for which a classical extraction module exists [Miq09a] and we cannot afford ourselves to use theorems or axioms we do not know how to realize. This means we cannot directly use Kőnig's lemma in our proof, unless we prove it before. Its most standard proof goes by contradiction and proves the equivalence between being an infinite tree and having an infinite branch. Formalizing this proof requires to have a representation of infinite trees in Coq which we would rather avoid. The solution we employ is to avoid a direct use of Kőnig's lemma but rather "inline" its proof to reach the contradiction.

3.1 The idea of our proof

The proof uses *reductio ad absurdum* to emulate Kőnig's lemma. We will assume inconsistency of the theory U and the absence of Herbrand trees for U. With these two hypotheses, we will show that any partial interpretation (of the atoms) consistent with U can be extended to a bigger partial interpretation still consistent with U. By iterating this process from the empty partial interpretation (which is obviously consistent with U), we will build incrementally a model of U, thus contradicting our hypothesis of inconsistency.

3.2 Design choices

Recalling the transformation of section 2.1, we see that there are cross-dependencies between on the one hand (user-provided) abstract data types—`atom` and `index`—and hypotheses—the inconsistency proof—and on the other hand the data types—`compound`, *etc.*—built by the proof. Regarding extraction, since we do not want to parametrize every proof by all these arguments, we use Coq typeclasses. They allow almost transparent usage of the abstract data types through the `Context` construction. They also ease the representation of several Boolean equalities and orders by allowing overloading and make the final theorem very easy to use since appropriate instances are automatically found whenever they exist.

3.3 The structure of the proof

The overall architecture of the proof is reflected in the file dependencies of the Coq development shown on Fig. 1. The first two files `Common` and `Optioned_Bool` are preliminary files and will not be discussed. Similarly, the file `order_N` is not part of the proof itself but rather consists of tools to help users create the required data types `atom` and `index`. The dashed edge represents a false dependency that exists only to build the instance of ThType (*cf.* Section 3.5) at once.

Optioned_Bool Some useful functions on the type `option bool`

Common Common tactics and some properties over relations

Orders Definition of the decidable equality and orders classes

Definitions Definitions of all data types: atom, index, compound, path, tree

Valuations Evaluation of a compound in a partial or total interpretation

iA_ordering Definition of the order on the dependent pairs $\langle i, a \rangle$

good_bad The actual proof

order_N User tools to build instances of the order classes from a countable set

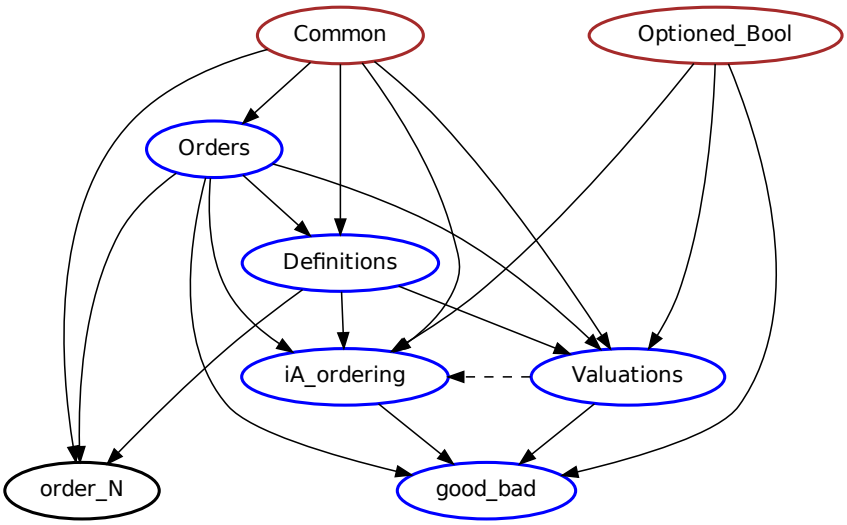


Figure 1: The Coq development architecture

3.4 Decidable orders and equalities

The proof requires decidable equality and ordering tests over several abstract data types, like `atom` or `index`, in order to effectively compare them. The following classes will each extend the previous one, adding extra properties that will be necessary for some types. For efficient computation, all tests are implemented as Boolean functions with a correctness proof. Note that it is not equivalent to use rich types containing specifications because in classical extraction such annotations are not removed, contrary to intuitionistic realizability (*cf.* Section 5.2).

```
Class DecEq (T : Type) := {
  eq : T → T → bool;
  eq_correct : ∀a₁ a₂, eq a₁ a₂ = true ⟷ a₁ = a₂}.
Notation "A ≡B" := (eq A B = true) (at level 40).
Notation "A ≢B" := (eq A B = false) (at level 40).
Notation "A ≡? B" := (eq A B) (at level 39).
```

This first class simply defines a decidable (Boolean) Leibniz equality together with some notations for easier use.

```
Class DecOrd {T : Type} `{Eq : DecEqual T} := {
 lt : T → T → bool;
 lt_irrefl : ∀a, lt a a = false;
 lt_trans : ∀a₁ a₂ a₃, lt a₁ a₂ = true → lt a₂ a₃ = true → lt a₁ a₃ = true}.
Notation "A ≪ B" := (lt A B = true) (at level 40).
Notation "A ⋘ B" := (lt A B = false) (at level 40).
Notation "A ≪? B" := (lt A B) (at level 39).
```

This second class defines a decidable strict order on types which already have a decidable equality. We introduce some extra notations to match mathematical habits.

```
Class TDWFOrd {T : Type} `{DecOrder T} := {
 dec : ∀a₁ a₂ : T, {a₁ ≪ a₂} + {a₁ = a₂} + {a₂ ≪ a₁};
 wf : ∀(P : Pred T), (∀ x, (∀ y, y ≪ x → P y) → P x) → ∀x, P x}.
```

where `Pred T` is the type `T → Prop` of predicates over `T`. This third class adds totality and well-foundedness to the previous order class.

```
Class MinPredOrder {T : Type} `{TDWFOrder T} := {
 minimum : T;
 minimum_OK : ∀t, minimum ≤ t;
 pred : T → T;
 pred_is_less : ∀t, (pred t) ≤ t;
 pred_is_sup : ∀a₁ a₂, a₁ ≪ a₂ → a₁ ≤ (pred a₂);
 pred_is_id : ∀a, (pred a) ≡ a → a = minimum}.
```

This last class defines a decidable total well-founded order with a predecessor function, which basically amounts to being isomorphic to natural numbers. Since there will be only one instance of this class, namely the `index` data type, and since we will also need a minimum on this type, for the sake of convenience we add it explicitly here although it could already be defined in the `TDWFOrd` class.

Each of these classes comes with an extended twin that contains proofs of useful common properties. An instance automatically builds the extended version from the basic one. For instance, the `TDWFOrd` class extends into a class `TDWFOrder` asserting some equivalences and the minimum principle.

```
Class TDWFOrder {T : Type} `{TDWFOrd T} := {
 lt_nleq : ∀a₁ a₂, a₁ ≪ a₂ ⟷ a₂ ⋦ a₁;
 nlt_leq : ∀a₁ a₂, a₁ ⋘ a₂ ⟷ a₂ ≤ a₁;
 min_princ : ∀(P : Pred T), (∃ a, P a) → (∃ a, P a ∧ ∀b, P b → a ≤ b)}.
```

This mechanism is used to define a non-strict order from an equality and a strict order (thus extending the `DecOrd` class) and prove its usual properties. Of course, it also comes with its own notations.

3.5 The abstract data types

Having all the tools we need to start, we first describe the hypotheses of Herbrand's theorem (*i.e.* the user input) as classes. We encounter again the different data types we discussed in section 2.1.

```
Class Atom := {
 atom : Set;
 AtomEq :> DecEq atom}.
```

Atom is an abstract data type representing the atomic formulæ of our theory, on which we can test equalities so that during evaluation of an atom, we can effectively test whether it appears or not in a partial interpretation.

```
Class Index := {
 index : Set;
 IndexEq :> DecEq index;
 IndexOrd :> DecOrd (E := IndexEq);
 IndexTDWFOrd :> TDWFOrd (E := IndexOrd);
 IndexMPOrder :> MinPrevOrder (HO := IndexTDWFOrd);
 lt_unbounded : ∀i₁ : index, ∃i₂, i₁ ≪ i₂}.
```

The index data type represents the parameters of the universal theory we consider. As will appear later in the proof (*cf.* Section 3.7), we need an decidable order and a predecessor for this type, hence we need it to be an instance of `MinPredOrder`. The final requirement `lt_unbounded` is for the sake of simplicity. Were it wrong, there would only be a finite number of indices, thus a finite number of atoms and one could represent all possible interpretations in a finite tree, rendering Herbrand's theorem useless.

```
Class ThType := {
 Th : index → compound;
 Th_absurd : ∀val, ¬(∀ i, eval val (Th i))}.
```

Finally, we reach the real hypothesis of our theorem: the theory and its proof of inconsistency. We recall that `compound` is the data type representing quantifier-free formulæ. It should be noted that this representation encompasses slightly more than universal formulæ since we do not have any requirement of uniformity for Th. Nevertheless, the Herbrand theorem we prove is equivalent to the usual one.

3.6 Internal data types: interpretations

Let us first introduce a data type we have already used, `compound`, representing quantifier-free formulæ. It is simply the Boolean algebra generated by the atoms.

```
Inductive compound `{Atom} : Set :=
 | Atomic (a : atom)
 | And (c1 c2 : compound)
 | Or (c1 c2 : compound)
 | Not (c : compound).
```

With this type, we can build the most important data type: partial interpretations.

```
Inductive path : Set :=
 | Top
 | Left (a : atom) (p : path)
 | Right (a : atom) (p : path).
```

It represents both partial interpretations and (reverse) paths in a Herbrand tree. Its `Left` constructor indicates that the atomic formula denoted by the atom `a` is true in this partial interpretation. Conversely, the `Right` constructor matches false atoms and `Top` denotes the empty partial interpretation. Containing only finite objects, this type naturally comes with a decidable equality (structural equality is decidable because atom equality is) and we can partially order it by extension, thus it is an instance of the `DecOrd` class. Using its equality, we build partial evaluation functions which evaluate atoms and compounds.

```
Fixpoint find : path → atom → option bool.
Fixpoint peval : path → compound → option bool.
```

The first function is exactly partial interpretation on atomic formulæ whereas the second one is its straightforward extension to quantifier-free formulæ. Their results may be undefined when a required atom is not in the partial interpretation denoted by the path, hence the use of the type `option bool`.

The last data type is of anecdotal importance and will be used only at the very end of the proof. It is a straightforward tree that will be used to build the final Herbrand tree once we know it exists.

```
Inductive Tree : Set :=
 | Contrad : index → Tree
 | Exp : atom → Tree → Tree → Tree.
```

3.7 Creating the order on dependent pair $\langle i, a\rangle$

Because compounds contain only a finite number of atoms, we can order the atoms appearing in one compound through a traversal of the given compound. By definition this order is total, and being finite it is also decidable and well-founded, *i.e.* an instance of the `TDWFOrd` class. Combining it lexicographically with the order on `index`, we get a total decidable well-founded order over the dependent pairs (i, a) with a belonging to $\mathrm{Th}\, i$. In what follows, we denote such dependent pairs by $\langle i, a\rangle$, omitting the side-condition $a \in \mathrm{Th}\, i$. Since `index` has a predecessor function and since a compound contains a finite number of atoms, we can define a predecessor function on the set of pairs $\langle i, a\rangle$. Yet giving an explicit predecessor for a given pair $\langle i, a\rangle$ requires some work and is defined by cases:

- 1st case: there is a predecessor in $\mathrm{Th}\,i$. To check this, we:
  1. compute the list of all atoms appearing in $\mathrm{Th}\,i$ (it defines the order on $\mathrm{Th}\,i$)
  2. find the element following a in the list if it exists
- 2nd case: a is the minimal atom in $\mathrm{Th}\,i$ and i is not the minimum index. We take the maximal element of $\mathrm{Th}(\mathrm{pred}\,i)$.
- 3rd case: a is the minimal atom of $\mathrm{Th}\,i$ and i is the minimum index. In this case, the pair $\langle i, a\rangle$ is the minimum of our order. By convention, we return $\langle i, a\rangle$.

The motivation for creating this order is to have an order on atoms compatible with the one on indices. In addition, given $j \in$ `index` we have a small finite initial segment p_j of the order (in fact, the smallest possible) containing all atoms a of all pairs $\langle i, a\rangle$ whose index i is smaller than j.

3.8 The proof itself

The essence of the proof lies in the reflection [Bou97] of the existence and structure of a Herbrand sub-tree in the inductive predicate `good`:

```
Inductive good p : Prop :=
 | good_leaf : ∀i, peval p (Th i) = Some false → good p
 | good_node : ∀a, find p a = None → good (Left a p) → good (Right a p) → good p.
```

In substance, a term of type `good p` is the sub-tree at the position defined by `p` of a Herbrand tree for Th. Indeed, the first branch of this inductive predicate expresses that we are at a leaf, *i.e.* there exists a contradiction in the theory denoted by Th at the axiom $\mathrm{Th}\,i$. On the contrary, the second branch expresses that we are at an inner node, *i.e.* we can add an atom a to the current path (associated to either true or false) and these paths have Herbrand sub-trees. Furthermore, this tree contains the proof of its correctness as annotations at each node. From a proof of `good Top`, it will then be trivial to extract a Herbrand tree by dropping the proof annotations and prove its correctness, giving the conclusion of our formalized theorem: `∃t : tree, Htree Th t = true` where `Htree` is a decision function for the property "`t` is a Herbrand tree for `Th`".

Going by contradiction, our proof starts by negating this property for the empty path. Negating the branches of the inductive, `¬(good p)` is classically equivalent to the following conjunction of formulæ.

$$\big(\forall i, \mathrm{peval}\,p\,i\,(\mathrm{Th}\,i) \neq \mathrm{Some\,false}\big) \wedge \big(\forall a, \mathrm{find}\,p\,a \neq \mathrm{None} \vee \neg\,\mathrm{good}(\mathrm{Left}\,a\,p) \vee \neg\,\mathrm{good}(\mathrm{Right}\,a\,p)\big)$$

Note that this predicate implies in particular that p does not interpret any $\mathrm{Th}\,i$ as false. Since p is a finite path, there exists an atom a not appearing in p. For a reason that will appear later, let us take the minimal one. Beware that this operation is not innocent because it requires the minimum principle which is classical theorem. Now we have the following implication:

$$\neg\,\mathrm{good}\,p \Rightarrow \quad \exists a, \quad \begin{array}{l}\mathrm{find}\,p\,a = \mathrm{None} \wedge \big(\forall b, \mathrm{find}\,p\,b = \mathrm{None} \Rightarrow a \leq b\big) \wedge \\ \big(\neg\,\mathrm{good}(\mathrm{Left}\,a\,p) \vee \neg\,\mathrm{good}(\mathrm{Right}\,a\,p)\big)\end{array} \tag{1}$$

With this formulation, one can see that it is possible to extend the path p into $\mathrm{Left}\,a\,p$ or $\mathrm{Right}\,a\,p$ still satisfying $\neg\,\mathrm{good}$. What we have to do now is to iterate this construction to get an infinite branch. Yet, this process is not constructive because of the existential quantifier so we cannot iterate it directly: from a path we only obtain a path predicate and not a concrete path. Embedding paths into singleton path predicates, *i.e.* translating `p` into `fun p' ⇒ p' = p`, we have path predicates that are isomorphic to paths. Modifying the predicate $\neg\,\mathrm{good}$ and the formula 1 to accommodate this new representation, the extension can be written as a function `Rextend` from singleton path predicates to singleton path predicates. We can then iterate it starting from the empty path predicate `fun p ⇒ p = Top` to get an increasing sequence `u_chain` of path predicates, all satisfying the predicate $\neg\,\mathrm{good}$ because `Top` does by hypothesis. Finally, we take the union u of this increasing sequence to have the infinite branch we want.

```
Definition Rextend : Pred path → Pred path.
Definition u_chain : nat → Pred path.
Definition u : Pred path := fun p ⇒ ∃n, u_chain n p.
```

What remains to prove is that u defines a model of $U = \forall i : \texttt{index}, \mathrm{Th}\, i$. In fact, we do not know whether u contains all atoms because only the ones appearing in a $\mathrm{Th}\, i$ were considered, so we cannot even say that u is a complete interpretation since there is a whole bunch of atoms that u says nothing about. However, these atoms are not relevant to us and we can arbitrarily interpret them without modifying the interpretation of U. The reason for this is that, in the extension process, we always take the minimal element not already in the path, ensuring that we do not forget any necessary atom. Let u^* be an interpretation extending u. As u^* is a complete interpretation, it cannot satisfy all $\mathrm{Th}\, i$ at the same time since U has no model. Let i_0 be an index such that u^* interprets $\mathrm{Th}\, i_0$ as false. Then u also interprets $\mathrm{Th}\, i_0$ as false. But by definition of u, this means there exists an $n \in \mathbb{N}$ such that the singleton path predicate `u_chain n` interprets $\mathrm{Th}\, i_0$ as false. The only element of `u_chain n` is the n^{th} extension of the empty path which has length n. Therefore we have a finite prefix p of u which interprets $\mathrm{Th}\, i_0$ as false. But by construction, p satisfies $\neg$ good and hence cannot interpret any $\mathrm{Th}\, i$ as false, a contradiction.

4 The wrapper of Herbrand's theorem proof

The premise of the experimental effectiveness rule is $U \vdash \bot$ which is not the hypothesis of Herbrand's theorem we have chosen (which is $\forall \mathscr{M}, \mathscr{M} \not\models U$). The bridge between the two statement is simply the correctness theorem which precisely states that the former implies the latter. Nevertheless, what we want to achieve in this section is a complete automated procedure to transform a Coq proof of $U \vdash \bot$ into the hypothesis we use in Coq for Herbrand's theorem, thus skipping over the intermediate step of $\forall \mathscr{M}, \mathscr{M} \not\models U$ presented in section 2.1. Being finite and universal, U can be written $\bigwedge_{i=1}^{n} \forall x, C_i(x)$. In this formulation, we implicitly say that we make no assumption over the signature on which U is built. To express this in Coq, we need to generalize over all symbols of the signature, which gives the following statement:

$$\underbrace{\forall f \ldots \forall g}_{\text{function symbols}} \quad \underbrace{\forall P \ldots \forall Q}_{\text{predicate symbols}} \left(\Big(\bigwedge_{i=1}^{n} \underbrace{\forall x, C_i(x)}_{\text{axioms}} \Big) \Rightarrow \bot \right).$$

To remove these new quantifications without loss of generality, we interpret the function symbols in the syntax: we build the free algebra of closed terms, the *Herbrand universe*. When we introduce the `atom` data type (*cf.* Section 2.1), we replace both the predicate symbols and all atomic formulæ by the *Herbrand base*, *i.e.* all ground instances of the predicates. Substituting the finite family of axioms by the indexing data type `index` and the function $\mathrm{Th} : \texttt{index} \to \texttt{compound}$, we reach the precise statement of Herbrand's theorem formalized in Coq:

```
∀atom, ∀index, ∀Th,
  (∀ val : atom → Prop, ¬(∀ i : index, eval val (Th i))) →
  ∃t : tree, Htree Th t = true.
```

where `Htree` is a decision procedure testing whether a tree `t` is a Herbrand tree for a theory `Th`. Notice that this transformation is intuitionistic so that any use of classical logic in the extracted realizer comes either from the proof of contradiction of the universal theory U or from the proof of Herbrand's theorem but not from the intermediate wrappers.

5 Classical extraction

The extraction of a Herbrand tree from a classical proof of its existence formalized in Coq is achieved in two stages as depicted by the following diagram.

$$\text{proof } M \xrightarrow{\text{adequacy}} \text{realizer } M^* \xrightarrow{\text{witness extraction}} \text{program } M^*\, T$$

where T is a suitable post-wrapper.

The first stage of the extraction process consists in translating the proof-term M of the Σ_1^0-formula `∃t : tree, Htree Th t = true` expressed in the calculus of inductive constructions (CIC [CDT10]) into

a *classical realizer* M^* (of the same formula) expressed in Krivine's λ_c-calculus. The correctness of this translation relies on the property of adequacy established in [Miq07], and its main interest is that it is able to deal with classical axioms such as the law of excluded middle (in `Prop`) or the axiom of proof-irrelevance.

The second stage consists in applying the classical λ-term M^* to a post-wrapper T whose aim is to turn the realizer M^* into a program M^*T that computes the Herbrand tree effectively, using the Σ_1^0-extraction technique described in [Miq10]. Evaluating the λ_c-term M^*T in Krivine's Abstract Machine then produces the desired tree.

Let us now consider the different ingredients of the extraction process more precisely.

5.1 λ_c-calculus: syntax and evaluation

The λ_c-calculus is the programming language to which we will extract the proof. The λ_c-calculus extends the pure λ-calculus [Chu41, Bar84] with the control instruction `callcc` ('call with current continuation') and continuation constants k_π [Kri09]. Unlike the pure λ-calculus, evaluation proceeds in the λ_c-calculus according to the call-by-name strategy, using Krivine's Abstract Machine (KAM).

Formally, the λ_c-language distinguishes three kinds of syntactic entities—*terms*, *stacks* and *processes*—that are mutually defined as follows:

$$\begin{array}{lllllllllll} \text{terms} & t,u & ::= & x & | & t\,u & | & \lambda x.t & | & \texttt{callcc} & | & k_\pi & | & \ldots \\ \text{stacks} & \pi & ::= & \alpha & | & t\cdot\pi & & & & & & & & \\ \text{processes} & p & ::= & t\star\pi & & & & & & & & & & \end{array}$$

(The reader is referred to [Kri09, Miq10] for a more formal presentation of the language.)

Terms of the λ_c-calculus contain the usual constructions of the λ-calculus plus the `callcc` instruction, reified stacks k_π (saved by `callcc`) and possibly many more instructions (*i.e.* term constants). Stacks (*i.e.* evaluation contexts) are finite lists of closed terms ended with specific stack bottoms α. In this paper, we shall consider only one stack bottom `nil`, so that stacks are exactly finite lists of terms. Finally, a process is simply a term put against a stack, ready for evaluation.

One of the main features of the λ_c-calculus is that it can be freely enriched with extra instructions coming with their own evaluation rules. An example of such an extra-instruction is the instruction `quote` that can be used to realize the axiom of dependent choices [Kri03]. For this reason, the relation of evaluation of the λ_c-calculus is not *defined*, but *axiomatized* with the following four rules:

$$\begin{array}{lrcl} \text{(Grab)} & \lambda x.\,t \star u\cdot\pi & \succ & t[u/x]\star\pi \\ \text{(Push)} & t\,u\star\pi & \succ & t\star u\cdot\pi \\ \text{(Save)} & \texttt{callcc}\star t\cdot\pi & \succ & t\star k_\pi\cdot\pi \\ \text{(Restore)} & k_\pi\star t\cdot\pi' & \succ & t\star\pi \end{array}$$

The rules (Grab) and (Push) describe the evaluation of pure λ-terms according to the call-by-name strategy, whereas the rules (Save) and (Restore) describe the save-and-restore mechanism performed by the instruction `callcc` and continuation constants k_π.

The main results of the theory of classical realizability (such as the property of adequacy [Kri09, Miq07]) are not tied to a particular relation of evaluation, but they more generally hold for *any* relation of evaluation that fulfills the above four axioms. In some situations it is desirable to consider extra axioms describing the computational behavior of extra instructions, that can be used either to realize formulæ that could not be realized otherwise (see [Kri03] for instance), or simply to provide more efficient versions of realizers. (A typical example (*cf.* Section 5.4) is the introduction of instructions for manipulating primitive integers such as described in [Miq10].)

5.2 Extraction function

The first stage of the extraction process consists in translating any CIC proof-term M of a proposition A into a λ_c-term M^* that realizes the formula A in the suitable realizability model. The adequacy of this translation is justified [Miq07] by a classical realizability model that extends Krivine's classical realizability model for classical second-order Peano Arithmetic [Kri09] to the calculus of constructions with universes enriched with classical reasoning at the level of the sort `Prop` of propositions.

Basically, the extraction function $M \mapsto M^*$ extracts the computationally relevant parts of the classical proof-term M. This translation is trivial on the constructions of the λ-calculus: variables are translated as themselves, as well as application and abstraction (removing the type annotation in the latter case). On the other hand, all types are collapsed to an inert constant written `.type`, thus reflecting the fact that types are computationally irrelevant, in the sense that they cannot appear in head position during the evaluation of a proof. (Types are only important at the logical level, where they play the role of specifications.)

In this setting, the principles of classical logic are deduced from the law of Peirce, which is itself translated as the instruction `callcc`. For instance, the law of excluded middle can be deduced in Coq from the law of Peirce by the following proof-term:

```
Definition excl_mid : ∀P, P ∨ ¬P :=
  fun P ⇒ Peirce (P ∨ ¬P)
     (fun k ⇒ or_intror P (¬P) (fun p ⇒ k (or_introl P (¬P) p))).
```

Through the extraction process, this proof term becomes the following λ_c-term

```
Define excl_mid  =
  λ_. callcc (λk. or_intror* .type .type (λp. k (or_introl* .type .type p))) ;;
```

whith `or_intror`* and `or_introl`* the λ_c-terms extracted from the constructors `or_intror` and `or_introl`, respectively. Notice that Peirce's law is translated as `callcc`, while the dummy constant `.type` is inserted at each place where a type is expected.

The extraction function $M \mapsto M^*$ used in this work is also extended to inductively defined data structures, pattern matching and functions defined by `Fixpoint`. The adequacy of the extended extraction function $M \mapsto M^*$ is justified by an extension of the realizability model presented in [Miq07] to the CIC, an extension that we shall not describe here. (Actually, it is not necessary to interpret the full mechanism of inductive definition of CIC, but only the constructions pertaining to the inductively defined type families that are used in the actual proof.)

Technically, inductively defined data structures are translated using standard second-order encodings. For instance, the three constructors of the following inductive definition

```
Inductive foo p1 p2 :=
 | C1 : foo p1 p2
 | C2 a : foo p1 p2
 | C3 b1 b2 b3 : foo p1 p2
```

are translated into the following λ_c-terms:

```
Define C1  = λp1 λp2              λe1 λe2 λe3 e1  ;;
Define C2  = λp1 λp2 λa           λe1 λe2 λe3 e2 a  ;;
Define C3  = λp1 λp2 λb1 λb2 λb3  λe1 λe2 λe3 e3 b1 b2 b3  ;;
```

(using the syntax of the Jivaro head reduction machine [Miq09c]). As we can see, these terms take the parameters of the inductive, the arguments of their constructor (if any), the eliminators for all constructors and they use the eliminator matching their constructor on their arguments. Thus, a term of type `foo` $p_1 p_2$ can be understood as a case analyzer, so that pattern matching on an inductively defined data structure can be simply translated as an application.

5.3 Witness extraction

One of the main properties of the classical realizability model described in [Miq07] (as well as of its extension to the CIC) is that its second-order fragment is isomorphic to the classical realizability model of classical second-order Peano arithmetic such as originally defined by Krivine [Kri09]. For this reason, all the witness extraction techniques described in [Miq10]—which only apply to realizers of arithmetic formulæ—automatically extend to all realizers coming from proof-terms in CIC via the extraction function $M \mapsto M^*$.

In order to extract Herbrand trees, we use the Σ_1^0-witness extraction technique [Miq10], that consists in taking a classical realizer M^* of a Σ_1^0-formula `∃x : D, f x = 0` and applying it to a suitable post-wrapper T whose aim is to extract the witness hidden in the classical realizer M^*. In practice, the λ_c-term T is simply defined by

$$T \quad \equiv \quad \texttt{S } (\lambda x \lambda y \texttt{ y } (\texttt{stop x}))$$

where `stop` is an instruction that retrieves the result and aborts computation, and where `S` is a *storage operator* [Kri94] whose implementation only depends on the representation of objects of type `D` in the λ_c-calculus.

Storage operators can be understood as a technique to force a call-by-value evaluation in the call-by-name setting of the KAM. Their idea is to decompose the term in order to force its evaluation and rebuild it as a value. More precisely, they extend a function defined on values to a function defined on all terms reducing to a previously accepted value. For instance, the storage operator for our type `tree` is

```
Define Mtree f t  =
 (t
   (Mindex (λi f (Trees.Contrad i)))
   (Matom (λa Mtree (λt1 Mtree (λt2 f (Trees.Exp a t1 t2)))))) ;;
```

It depends on two other realizers `Mindex` and `Matom` for the types `index` and `atom` respectively that are omitted here but follow a similar pattern.

5.4 Realizer optimization

By definition, extracted terms follow the structure of the proofs they come from. Although correct, this sometimes turns out to be completely inefficient. For instance, Krivine's classical realizability interprets true equalities between natural numbers as the identity. This implies that proofs of equalities are computationally equivalent to the identity. Yet, the terms extracted from these may be much more involved, using for instance induction and pattern-matching, meaning that the realizers of this kind of proofs destruct their argument before constructing it back, which obviously leads to inefficient computation.

The idea of realizer optimization is to replace some of the extracted realizers by more efficient ones. However, we must ensure that the new realizer has the same computational content as the old one, *i.e.* that it also realizes the theorem satisfied by the old realizer. This simple mechanism provides complexity improvements at the scale of the order of magnitude because it can replace functions using unnecessary recursive calls (thus linear functions) to constant-time functions. Experimentation on the White Crow theory confirms these results: we move from a quadratic time-complexity to a linear one.

A deeper but heavier optimization technique also exists: changing the representation of one data type into a more efficient one (in the sense of time- and/or space-complexities). It requires to modify all constructors and destructors of the old data type to produce and use optimized data but also all re-implement the functions using or returning objects of this data type to benefit from the improvement. Finally, we need realizers of the logical implications between the two data types which will compute the conversions between the two representations.

The best example of such optimization is natural numbers: in Coq they are inductively defined as unary integers (according to Peano's axioms) whereas we want to use the primitive binary representation. If we only change the constructors (*i.e.* the constant 0 and the successor function) and the destructor (*i.e.* pattern-matching), we will have integers stored as binary words but they will still be used as unary integers, *e.g.* with a recursive definition for addition. Therefore, we also need to replace all arithmetical operations by their binary counterparts in order to enjoy a time improvement. The interested reader can have a look at [Miq09b, pp. 95–98] for more information.

6 Another solution: direct implementation

6.1 Intuitive presentation

Instead of extracting Herbrand's theorem, there is a more direct way to build Herbrand trees. This solution has already been presented and proved correct in [Miq09b, pp. 98–104], so here we shall focus more on its intuition and implementation. Although faster, the implementation of this solution is not certified, contrary to the extraction method.

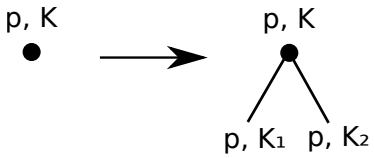
Informally, the computational meaning of the realizer of the contradiction proof of U

$$U \vdash \bot \qquad i.e. \qquad \bigwedge_{i=1}^{n} \forall x_1 \dots \forall x_{k_i} C_i(x_1, \dots, x_{k_i}) \vdash \bot$$

is to use realizers of the axioms $\forall x_1 \ldots \forall x_{k_i} C_i(x_1, \ldots, x_{k_i})$ to find a contradiction. The realizers of theses axioms themselves depend only on the realizers of the atomic formulæ. (Realizers of the logical connectives have a known computational meaning.) Thus, given realizers for the atomic formulæ appearing in U, we can find a contradiction for U. But the realizers of atomic formulæ depend only on the truth of the atoms, so that if we interpret atoms as equalities, there are as many set of realizers for the atoms as interpretations. Thus, by changing the realizers of atomic formulæ, we change the interpretation, hence we change the construction of the contradiction.

The overall idea of this method is to evaluate directly the realizer of the contradiction proof and try several realizers for the atomic formulæ (*i.e.* several interpretations) while doing so. Indeed, with one set of realizers for the atoms we only get one interpretation and thus only one path of our tree. So if we try different sets of realizers, we will eventually cover all branches of the tree we want to build.

In order to do this, we add a scheduler on top of the KAM. It will fork the currently evaluated process each time a new atomic formula is encountered and give to the new threads the two different realizers for this atom. Doing this for every atom the contradiction proof encounters, we will cover the relevant part of all interpretations. More precisely, each process now has a local knowledge base K associating truth values (or the corresponding realizers) to atoms. When evaluation reaches the realizer of an atom a not present in K, we duplicate the current process, adding a associated to true to the knowledge base of one branch and associated to false in the other.



where $K_1 = K \cup \{(a, true)\}$ and $K_2 = K \cup \{(a, false)\}$. If the atom is already present in K, we simply put in head position the realizer corresponding to its truth value in K.

In this regard, atom realizers can be seen as "system calls" that wake up the scheduler, fork the current thread, modify the knowledge base of both new threads and run them in parallel. The built tree is exactly the execution tree of the realizer of the contradiction proof when we consider all possible realizers for the atoms, *i.e.* the execution tree when we consider all interpretations, with extra labels on the leaves for the counter-example, that is a Herbrand tree. See Fig. 2 for the example of the White Crow theory.

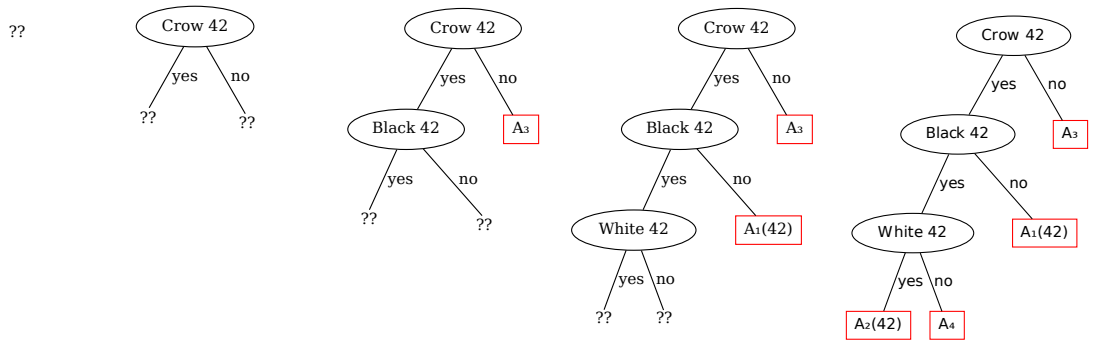


Figure 2: The five construction steps for one White Crow Herbrand tree

6.2 Current implementation

First of all, we need determinism in order to keep the simple extraction function described in section 5.2, so we will have to explicitly fork and schedule all the sub-processes that will appear. The current implementation uses a modified version of the λ_c head reduction machine Jivaro [Miq09c] adding two global stores `zipper` and `cont` that survive backtracks. The first one contains a zipper [Hue97] of the (partial) Herbrand tree being built and the second one contains the continuation of the construction of the tree. If the inner nodes of the zipped tree are labeled by atoms (just like a Herbrand tree), its leaves

are threefold: either a contradiction (as in a standard Herbrand tree), a frozen process (some computation that remains to be done) or the working node to which is attached the process being evaluated.

The reason why we need the `cont` store is simply that evaluation ends a branch when we reach a realizer of $\bot$, *i.e.* an inconsistent state in which there can be anything on the stack and all future computation are meaningless. As long as the tree is not complete, there is still an incomplete branch, so we can switch evaluation to this pending process. When we complete the tree, we are in an inconsistent state but have no other process to switch to, so we need to restore a continuation previously stored, hence the existence of the `cont` store.

In this framework, the knowledge base is the reverse path connecting the current node (*i.e.* the working node) to the root of the tree. It contains all atoms that were previously evaluated before reaching the current process, so the position of the current process in the tree gives its knowledge base if we see the tree as a BDD.

We add five new instructions to the terms of the λ_c-calculus: `test`, `contradict`, `reset`, `finish` and `save`, together with the following new reduction rules.

$$\begin{array}{rcl}
\texttt{test} \star a \cdot u_1 \cdot u_2 \cdot \pi & \succ & u_1 \star \pi \\
\texttt{test} \star a \cdot u_1 \cdot u_2 \cdot \pi & \succ & u_2 \star \pi \\
\texttt{contradict} \star t \cdot \pi & \succ & u' \star \pi' \\
\texttt{reset} \star k \cdot \pi & \succ & k \star \pi \\
\texttt{finish} \star \pi & \succ & c \star t \cdot \pi \\
\texttt{save} \star c \cdot k \cdot \pi & \succ & k \star \pi
\end{array}$$

We can see that the intuitive meaning of these new rules (notably forking the current process) does not appear here. Furthermore, the first two rules seem concurrent but are not: they will occur in disjoint circumstances. The reason for this is that the intended behavior of these rule is (partly) external to the KAM, so their interest lies in their side effects, listed below, which are completely transparent to the evaluation relation.

- 1st rule: no side effect but occurs only when the atom a is present in the current knowledge base and is associated to *true*;
- 2nd rule:
  - either the atom a is already evaluated to false inside the current knowledge base and there is no side effect;
  - or the atom a has never been encounter before. In this case, we extend the tree by replacing the current working node by `Node(`a`, Frozen` (u_1, π)`, Working_Node)`, expressing that we just made a decision for the realizer of a and that we continue on the false branch having previously stored the process (u_1, π) corresponding to the true branch;
- 3rd rule: ends the current branch with a contradiction labeled by t and switch to another pending thread $u' \star \pi'$ (*i.e.* a leaf `Frozen`(u', π') in the zipped tree) if there is one, otherwise calls `finish` π;
- 4th rule: empties the content of the `zipper` store;
- 5th rule: applies the content of the `cont` store to the zipped tree;
- 6th rule: put c into the `cont` store.

The overall behavior of such a program is to build the Herbrand tree right-to-left (because we first choose the false sub-tree) by extending the current branch until reaching a contradiction. It then switches to the next branch and repeat this process till completion of the tree. The termination is ensured by the adequacy theorem of classical realizability because the executed term is extracted from a Coq proof (see [Miq09b] for more details).

6.3 Concrete usage of the program

We run the program in the modified KAM by applying the realizer extracted from the contradiction proof of the theory U to realizers of the axioms of U, automatically generated from their Coq statements. Thus, a common λ_c-term for computing a Herbrand tree of the White Crow theory is the following:

```
Define eval_tree proof k  = save (Mtree k) proof  ;;
Define crow_test  =
 eval_tree
   (Top.crow_Th .type .type .type Top.CB Top.nBW Top.C42 Top.W42)
   print  ;;
```

where `Top.crow_Th` is the realizer extracted from the contradiction proof, `Mtree` is a storage operator for the tree data type and `Top.CB`, `Top.nBW`, `Top.C42` and `Top.W42` are realizers of the four axioms of the White Crow theory (built automatically by a Coq tactic).

We have introduced a macro `eval_tree` for a complete transparent usage of the tree construction program. Since `crow_Th` is extracted from Coq, it requires three dummy parameters (the predicates Crow, Black and White of type `nat → Prop`) represented by the constant `.type`. The instruction `print` is a Jivaro primitive which prints its argument as a term and can be replaced by any retrieving term for the tree.

7 Future works

It would be interesting to understand what is the precise computational behavior of the extracted proof and see whether it is possible to remove the linear dependency (*cf.* Section 5.4) that appears in the White Crow example. It is probably connected to the choice of the order on indices and a clever order putting the counter-example index and atoms first would probably run much faster.

In the long run, developing a theory of realizer optimization is definitely worth the effort, considering the complexity improvements involved. In this sense, what could be an analog of Harrop formulæ, known to have no computational meaning in intuitionistic realizability, in Krivine's framework? The objective would be to have a wide-enough class of formulæ (including at least implications between atomic equalities) for which an optimized realizer could be automatically substituted during extraction based on the shape of the formula.

The idea of the second method has been proved correct in [Miq09b] but the current implementation which features explicit scheduling and no parallelism remains to be formally proved. It would also be interesting to know if we can embed it into standard classical realizability without a scheduler. For plain realizability, it is impossible since nothing can survive a backtrack but if we allow global stores, we expect it to be doable considering the current implementation.

References

[Bar84] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and The Foundations of Mathematics*. North-Holland, 1984.

[Bou97] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *TACS'97. Springer-Verlag LNCS 1281*, pages 515–529. Springer-Verlag, 1997.

[CDT10] The Coq Development Team. The Coq Proof Assistant Reference Manual – version v8.3. Technical report, INRIA, December 2010.

[Chu41] A. Church. *The calculi of lambda-conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton, 1941.

[Gri90] Timothy Griffin. A formulae-as-types notion of control. In *POPL*, pages 47–58, 1990.

[Hue97] Gérard Huet. The zipper. *J. Funct. Program.*, 7:549–554, September 1997.

[Kri94] Jean-Louis Krivine. A general storage theorem for integers in call-by-name lambda-calculus. *Theoretical Computer Science*, 129:79–94, 1994.

[Kri03] J.-L. Krivine. Dependent choice, 'quote' and the clock. *Theor. Comput. Sci.*, 308(1-3):259–276, 2003.

[Kri09] Jean-Louis Krivine. Realizability in classical logic. In *Interactive models of computation and program behaviour*, volume 27 of *Panoramas et synthèses*, pages 197–229. Société Mathématique de France, 2009.

[Miq07] Alexandre Miquel. Classical program extraction in the calculus of constructions. In *Computer Science Logic (CSL)*, pages 313–327, 2007.

[Miq09a] Alexandre Miquel. *The classical extraction module for Coq*, 2009.
Available at `http://perso.ens-lyon.fr/alexandre.miquel/kextraction/`.

[Miq09b] Alexandre Miquel. De la formalisation des preuves à l'extraction de programmes. HdR thesis, Université Paris 7, December 2009.

[Miq09c] Alexandre Miquel. *The Jivaro head reduction machine for the λ_c-calculus*, 2009.
Available at `http://perso.ens-lyon.fr/alexandre.miquel/jivaro/`.

[Miq10] Alexandre Miquel. Existential witness extraction in classical realizability and via a negative translation. In *Logical Methods in Computer Science (LMCS)*, 2010. to appear.

[Miq11] Alexandre Miquel. The reasonnable effectiveness of mathematical proof. In Myriam Quatrini and Samuel Tronon, editors, *Anachronismes logiques*, Logique, Langage, Sciences, Philosophie. Publications de la Sorbonne, 2011.